

第5章 函数与变量

5.1 函数的定义与调用

5.2 程序的函数分解

5.3 循环与递归

5.4 外部变量与静态局部变量

5.5 声明与定义

5.6 预处理

5.7 程序动态除错方法（二）

5.4 外部变量与静态局部变量

从前文可见，程序中的函数与函数之间彼此是平等的，不能嵌套定义。由此可见，从全局角度来看，程序实际上是由一系列函数构成的。程序中有多个层次。

程序表层
(函数外部)

```
#include <iostream>
using namespace std;
```

函数

```
int isprime(int n) {
    for (int m=2 ; m * m <= n; m++)
        if (n % m == 0) return 0;
    return 1;
}
```

```
int main() {
```

```
    int n;
```

```
    for (n=2; n<=100; n++) {
```

```
        if (isprime(n)) cout << n << " ";
```

```
    }
```

```
    return 0;
```

```
}
```

复合结构

前文多次说到，在程序中的任何复合语句**内部**的任何位置都可以定义变量。那么，**在函数的外部**（即程序文件中的多层复合语句的最外层）**是否可以定义变量呢？**

C和C++语言规定，**在函数的外部也可以定义变量。**

把变量定义语句写在函数的外部，这样**以外部定义的形式所定义的变量**称为**外部变量**(external variable)或**全局变量**(global variable)。



在作用域和存在期这两方面，它们的性质与函数内部定义的变量（局部变量）有很大的不同。

外部变量的作用域是从其定义位置开始，到源文件结束之处。



【例5-21】对2-100之间的整数判断是否质数。

```
#include <iostream>
```

```
using namespace std;
```

```
int gn; //定义外部变量
```

```
int isprime2() {
```

```
    for (int m = 2 ; m * m <= gn; m++)
```

```
        if (gn % m == 0) return 0;
```

```
    return 1;
```

```
}
```

```
int main() {
```

```
    for (gn = 2; gn <= 100; gn++)
```

```
        if (isprime2())
```

```
            cout << gn << endl;
```

```
    return 0;
```

```
}
```

外部变量的作用域：
从其定义位置开始，
到源文件结束之处。

```
#include <iostream>
using namespace std;
```

```
int isprime2() {
    for (int m = 2 ; m * m <= gn; m++)
        if (gn% m == 0) return 0;
    return 1;
}
```

外部变量的作用域：
从其定义位置开始，
到源文件结束之处。

```
int gn; //定义外部变量

int main() {
    for (gn = 2; gn <= 100; gn++)
        if (isprime2())
            cout << gn << endl;
    return 0;
}
```

如果修改外部变量 gn 的定义位置，则上方的函数不能使用此变量。

外部变量的存在期又怎么样呢？



外部变量的存在期是程序的整个执行期间。

在程序执行开始时，所有外部变量都已有定义，在内存中被分配了存储空间。外部变量的这种有定义、被分配了存储空间的状态一直延续到程序结束，它们与对应存储位置的关联也保持不变。

请对比局部变量的存在期……

外部变量的优点与用途：外部变量定义在函数之外，一个函数可以把公共数据存入这种变量，另一函数就可以直接使用它。因此，全局变量可以看作函数间交换数据的一种通道，利用它们交换数据，程序写起来直截了当。后面章节里有这方面的例子。

使用外部变量的缺点：也应注意由于使用外部变量而引起的其它效果。如果一个函数的定义里访问了某个外部变量，那么这个函数就对该变量就有了依赖性。

例如上例中的isprime2已不再是独立的函数了，不能孤立地把它定义复制到另一个程序里使用。如果想在其它地方使用这个函数，函数的环境里必须有同样的外部变量。

因此，为了尽量使函数维持独立性，**通常在程序尽量不使用全局变量为好。**

为此，需要用户熟练使用函数传递数值的机制，合理地设计变量的类型。

- 一般说，在比较复杂的程序里，人们倾向于把**比较大、具有唯一出现，而且是在程序中被许多函数公用的数据对象（例如很大的数组）**定义为外部变量。
- 对于一般数据对象则采用参数方式传递。

对于一个具体问题应当怎样处理，要考虑软件系统实现的方便、清晰、数据安全等许多因素。选用合理的数据传递方式才能编写出功能完善、执行过程优美的好程序。

5.4.2 变量定义的嵌套

程序中的两种作用域：

- 每个复合语句确定了一个作用域：局部作用域；
- 整个程序：全局作用域。

局部定义（主要是指变量）的作用域是局部的；

全局作用域是所有外部定义（外部变量定义、外部函数定义等）的作用域。

多种作用域的存在造成了作用域的嵌套：（1）作为函数体的复合语句嵌套在全局作用域里；（2）复合语句里还可以有嵌套的复合语句。

问：在嵌套的作用域中是否可以使用同名变量？

这时仍然服从语言对同名变量的规定：当内层复合语句出现同名变量定义时，外层同名定义将被内层定义遮蔽。

也就是说，在使用该变量名时，实际上是优先使用内层定义的变量。

同样，当外部定义的全局变量与函数中的局部变量同名时，函数中的局部变量会遮蔽全局变量（在函数内使用该变量名时，优先使用局部变量）。

【例5-23】程序示例，说明出现同名变量定义时的变量名使用情况。

```
int n; //外部变量（全局变量）
```

```
int func (int m) {
```

```
    int n = 10; //n2
```

```
    cout << "n= " << n << endl; //n2
```

```
    for (int n = 1; n <= 8; n++) { //n3
```

```
        m = m + n * n; //n3
```

```
        cout << "n= " << n << " m= " << m << endl; //n3
```

```
    }
```

```
    cout << "n= " << n << endl; //n2
```

```
    return m;
```

```
}
```

```
int main() {
```

```
    n = 5;
```

```
    cout << "n= " << n << " func(n)= " << func(n) << endl;
```

```
    return 0;
```

```
}
```

虽然上面这段程序并不长，但是由于在嵌套的作用域中出现了同名变量，导致程序变得晦涩难懂。

在编程实践中，局部变量与全局变量同名极易导致程序编写者犯逻辑错误。

本书作者觉得对读者有用的建议是：在程序中使用合理的命名规则给变量命名，以尽量避免在嵌套的作用域中出现同名变量！

例如，对全局变量故意添加字母前缀“g”或“g_”（“g”字母是“global”的缩写），而且局部变量故意使用多个字母和数字来命名以避免同名。

例：

执行下列程序后输出的结果是

()

```
#include <iostream>
```

```
using namespace std;
```

```
int a = 3, b = 4;    //外部变量
```

```
void fun(int x1, int x2){
```

```
    cout << x1 + x2 << ", " << b << endl;
```

```
}
```

```
int main(){
```

```
    int a = 5, b = 6;    //局部变量
```

```
    fun(a, b);
```

```
    return 0;
```

```
}
```

内层作用域里的同名变量定义（在此局部）遮蔽外层同名变量定义。

A. 3, 4

B. 11, 1

C. 11, 4

D. 11, 6

5.4.3 静态局部变量

假设需要一个函数，调用时通常输出空格，调用第10次时输出换行符。取名 format，无参。

可用它改造打印完全平方数程序，使每行输出10个数：

```
for (n = 1; n * n <= 200; n++) {  
    cout << n * n;  
    format();  
}
```

许多地方都可能用它。

初看这个函数可能很简单。

```
void format(void) {  
    int m = 0;  
    if (++m == 10) {  
        putchar('\n'); m = 0;  
    } else  
        putchar(' ');  
}
```

这个函数无论调用多少次都不输出换行。

原因：m 是局部的自动变量，每次调用都重新建立并初始化为 0。无论调用 format 多少次，m 也不可能等于10。

为完成工作，需要在 format 两次调用之间传递信息。

自动变量存在期过短，无法完成这种传递。

解决办法之一：可采用外部变量，存在期长，能保持值不变，可在任何函数（包括format）里使用。

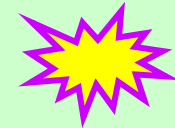
```
int m = 0;
void format(void) {
    if (++m == 10) {
        putchar('\n'); m = 0;
    }
    else putchar(' ');
} /* 功能正确 */
```

问题：程序其他部分可能不当心修改了 m（例如把局部变量 n 写成 m），导致计数错误。——这样使用全局变量不好！

信息隐蔽/保护很重要，小到程序的组织，大到关系国家安全的重要系统，都必须关注这个问题。

需要一种变量：

局部作用域，限制在函数内，防止越权访问；全程存在期和一次初始化，以便在函数调用之间传递信息，值在不同调用之间保持不变。



静态局部变量：定义在函数内部，前面加 **static**。

```
void format(void) {  
    static int m = 0;  
    if (++m == 10) {  
        putchar('\n'); m = 0;  
    } else  
        putchar(' ');  
}
```

需要局部作用域/全程存在期的变量，应使用static。

典型应用：随机数生成函数。

生成随机数需要前次递推值（种子值），用变量保存。

不能用自动变量，静态局部变量是一种合理选择。例：

```
int random () {  
    static unsigned long seed = 1;  
    return seed = (seed * 1103515245 + 12345) % 32768;  
}
```

无参函数头部：

int random() ... 或者 int random(void) ...

原型必须用 int random(void);

5.4.4 变量初始化

	定义位置	作用域	存在期
	函数内部（未加static） 局部变量	所在的复合结构，是 局部的	所在的复合语句的 一次执行。自动建 立和销毁：自动变 量
	函数外部：外部变量	从定义语句到程序结 束：全局变量	全程存在期
	函数内部（加static） 静态局部变量	所在的复合结构，是 局部的。	全程存在期

5.4.4 变量初始化

自动变量

- 每次进入作用域时建立，如果有初始化语句，就初始化。多次进入，则多次初始化。
- 初始化可用任何表达式，可包含函数调用、引用其他变量等。
- 函数参数是局部自动变量，在运行进入函数体前建立，用实参表达式的值初始化。

外部变量/静态局部变量

- 在程序执行前建立并初始化，只做一次（这是保证format函数能工作的一个因素）
- 初始化表达式只能用静态可求值的表达式，如文字量，文字量/符号常量/基本运算符构造的表达式。不能有赋值、增量减量运算等。

没有初始化的变量定义

变量定义可以没有初始化部分。如果没有，则：

- 自动变量若不初始化，将处于未初始化的状态，其值无法确定。
- 外部变量/静态局部变量自动初始化为0（默认初值）

自动变量未经初始化就使用，既不正确也不合理（初学者常犯的错误）。

一些编译器会给出警告。但有时未必都能给出警告，有时给出的警告未必正确。

【例5-25】 写一个程序，里面分别包含有初始化和无初始化的局部变量、外部变量和静态局部变量，对比它们的初始化性质。

```
#include <iostream>
```

```
using namespace std;
```

```
int ga = 10; //定义全局变量 ga 并初始化为10
```

```
int gb; //定义全局变量 gb ，不初始化
```

```
void func() {
```

```
    static int sta = 10; //定义静态局部变量 sta 并初始化为10
```

```
    static int stb; //定义静态局部变量 stb，不初始化
```

```
    int a = 10; //定义局部变量 a 并初始化为10
```

```
    int b; //定义局部变量 b，不初始化
```

```
    cout << a << '\t' << b << '\t' << ga << '\t' << gb << '\t'  
        << sta << '\t' << stb << endl;
```

```
    a += 100;    b += 100;    ga += 100;    gb += 100;
```

```
    sta += 100;    stb += 100;
```

```
}
```

```

int main() {
    cout << "a\t" << "b\t" << "ga\t" << "gb\t" << "sta\t" << "stb" << endl;
    func(); //第 1 次调用
    func(); //第 2 次调用
    func(); //第 3 次调用
    return 0;
}

```

局部变量多次初始化 外部变量仅初始化一次 静态局部变量仅初始化一次

a b		ga gb		sta stb	
10	4504194	10	0	10	0
10	4504294	110	100	110	100
10	4504394	210	200	210	200

局部变量未初始化则其值未定，
不会自动初始化

无初始化语句则自动初始化为0

5.4.5 名字空间

- 局部变量的作用域
- 外部变量和静态局部变量及其作用域
- 多文件开发时对外部变量的声明

至此，已经看到了 C/C++ 语言中所定义的3个层次的作用域，即文件(编译单元)、函数和复合语句。

除此之外，C++语言还引入了类作用域。在不同的作用域中可以定义相同名字的变量，互不干扰，系统能够区别它们。

通常，由于有这几种作用域的限定，已经足够解决程序中的同名冲突了。

但是，在更大的多文件开发工作中，上述几种作用域仍然不足以解决可能存在的同名冲突。例如：

(1) 一个较大的程序中需要使用多个函数库（包括编译系统提供的库、由软件开发商提供的库或者用户自己开发的库），为此需要包含有关的头文件，如果在这些库中包含有与程序的全局实体同名的实体，或者不同的库中有相同的实体名，则在编译时就会出现名字冲突。

(2) 一个大程序项目被分解为多个文件，交由多个编程人员分别编写，那么这些编程人员可能在各自负责的多个文件中使用上面几种作用域的规则解决变量名冲突，但是所有编程人员所编写的源代码凑在一起编译时，仍然可能出现变量同名冲突和函数名同名冲突。

为了避免这类问题的出现，人们提出了许多方法。但是这样的效果并不理想，而且增加了阅读程序的难度，可读性降低了。

ANSI C++ 标准中引入了名字空间(namespace)的概念，较好地解决了这个问题。

引入命名空间的目的是使由库的设计者命名的全局标识符能够和程序的全局实体名以及其他库的全局标识符区别开来。

它实际上就是一个由程序设计者命名的内存区域，程序设计者可以根据需要指定一些有名字的空间域，把一些全局实体分别放在各个命名空间中，从而与其他全局实体分隔开来。

例如，可以定义如下命名空间：

```
namespace ns1 { //定义命名空间ns1
    int a;
    double b;
}
```

其中，namespace 是定义命名空间所必须写的关键字，ns1 是用户自己指定的命名空间的名字（可以用任意的合法标识符），在花括号内是声明块，在其中声明的实体称为命名空间成员。现在这个命名空间的成员包括变量 a 和 b 。

注意 a 和 b 仍然是全局变量，仅仅是把它们限定在指定的命名空间中而已。如果在程序中要使用变量a和b，必须加上命名空间名和作用域分辨符 “::”，如ns1::a，ns1::b。这种用法称为**命名空间限定**(qualified)，这些名字称为被限定名(qualified name)。

程序设计人员可以根据需要设置许多个命名空间（不能同名），每个命名空间名代表一个不同的命名空间域。这样，通过使用命名空间，就可以建立起一些互相分隔的作用域，把不同的库中的全局实体（变量、函数、.....）放到不同的命名空间中进行定义（或声明），或者说，用不同的命名空间把不同的实体隐蔽起来，彼此分隔开来，从而可以有效地避免了同名冲突。

标准 C++ 库的所有的标识符都是在一个名为 std 的命名空间中定义的，或者说标准头文件(如iostream)中函数、类、对象和类模板是在命名空间 std 中定义的。

在程序中用到C++标准库中的实体时，需要使用std作为限定。例如当我们需要使用cout对象进行输出时，可以写：

```
std::cout << "Hello, world!" << endl;
```

显然，如果程序中每一处都这样写，就显得颇为罗索。为了使编程者能方便地使用某个命名空间中的所有程序实体，C++ 提供了using namespace语句。这种语句的一般格式为

`using namespace 命名空间名;`

因此，为了使用标准C++库的所有的标识符，我们只需要在程序头部写如下语句：

`using namespace std;`

这样就在程序中声明了在本作用域中要用到命名空间 std 中的成员，在使用该命名空间的任何成员时都不必用命名空间限定。——正因为如此，我们才可以在程序中方便地使用“cout <<”和“cin >>”进行输出和输入。

第5章 函数与变量

5.1 函数的定义与调用

5.2 程序的函数分解

5.3 循环与递归

5.4 外部变量与静态局部变量

5.5 声明与定义

5.6 预处理

5.7 程序动态除错方法（二）

5.5 声明与定义

在前文的各种示例程序中，读者可能会得到一个印象：一个源程序里可以包含多个函数，这些函数需要全部写在一个文件中；或者说，每一个程序的所有内容都需要写在一个文件中。

这种理解是不正确的。把每个小程序的所有内容写在一个文件中是合适的，但是在开发大型程序时，程序中所需要的函数很多，而且可能需要多个编程者协同工作，这时把程序中的所有内容写在一个文件中就不方便了。

实际上，系统支持把一个程序拆分为多个文件进行开发，以方便多个编程者协同工作。当然，此时就需要一些特定的机制帮助进行这样的多文件开发工作。

需要了解：（1）函数和变量的声明与定义；（2）预处理命令中的文件包含功能。

5.5.1 先定义后使用

在程序里，每个有名字的程序对象（变量、函数都是程序对象）都有**定义点**和**使用点**。一般说，一个对象只应有一个定义点，可以有多处使用点。为保证使用与定义的一致性，通行的规则是应当“**先定义后使用**”。通常总是这样做：

（1）所有变量都要求它们的**定义出现在使用变量的语句之前**，这就保证了它们的先定义后使用。

（2）**把被调用的自定义函数写在主调函数之前**，这样就保证了被调用的函数先定义后使用。

规定“先定义后使用”，是因为程序对象的使用方式依赖于它们的性质。如果没有定义在先，就难以知道使用是否正确。为保证语言系统能正确处理程序，基本原则是：**保证从每个对象的每个使用点（调用处）向上看，都能得到与正确使用该对象有关的完备信息。**

对于变量，在每个使用点向上看，都应该能得到该变量的完备信息：变量类型和变量名。从而知道该如何正确地使用该变量。

对于函数，在每个调用处向前看，都应该能看到该函数的完备信息：函数返回值类型、函数名、参数个数和类型。在函数调用处需要检查参数个数是否正确，各参数的类型是否与函数定义一致，如果不一致能否转换（必要时插入转换动作）等。由于返回值可能参加进一步计算，也要做类似处理。如果向上看不到函数的类型特征，就无法正确完成这些检查和处理。

在函数的递归定义中，这一要求仍能满足。例如：

```
long fact(long n) {  
    return n <= 0 ? 1 : n * fact(n - 1);  
}
```

在函数体里的递归调用点向上能看到函数头部描述的类型特征。

5.5.2 定义与声明

但是，在更复杂的程序设计中，也存在一些情况，无法通过上述“先定义后使用”的方式安排变量和函数的定义位置的来解决调用点与使用点间的信息交流关系。例如：

- (1) 程序中有两个函数互相调用对方
- (2) 在多文件开发中需要把多个全局变量和多个函数分开在多个文件中进行编写，并对这些文件进行分别编译、最终对整个程序进行编译。
- (3) 在多文件开发中，如果把使用到同一个全局变量的多个函数划分在不同文件中，然后在每个文件中都定义相同名字的变量，那么在单独编译这些文件时是正常的，但最终对整个程序进行编译时就会导致变量名冲突。

为了解决这些问题，需要我们准确地理清“定义”
(英文动词为 define，名词为 definition) 的含义：

对**变量**的“**定义**”是用于为变量分配存储空间，还可为变量指定初始值。在程序中，一个变量有且仅有一个定义。

对**函数**的“**定义**”是指对函数功能的确立，包括指定函数名，函数值类型、形参类型、函数体（函数内部需要执行的计算）等，它是一个完整的、独立的函数单位。如果函数内部有静态局部变量，那么程序在启动执行时就会为该函数分配相应的变量存储空间。

程序编译时，**编译器总是从上往下对程序文件进行扫描。**

因此在扫描到“先定义后使用”方式所安排的变量定义语句和函数定义语句块时，实际上不仅理解了这些程序对象的定义，同时还理解了它们的性质和特征，以便在程序中遇到使用这些程序对象的语句时，对它们进行对照检查。

“先定义后使用”方式所安排的变量定义语句和函数定义语句块实际上是同时做了两件事：

- (1) **定义**了程序对象，
- (2) 向编译器**声明**这些程序对象的性质和特征。

这种定义方式也称为“**定义性声明(defining declaration)**”，对于小型程序是简便合理的，也是初学者通常所做的。

在更复杂的程序设计中，存在一些情况，无法通过这种“先定义后使用”的方式安排变量和函数的定义位置的来解决调用点与使用点间的信息交流关系。为了解决这些问题，就需要把“定义”和“声明”分离开来进行处理，即在程序文件中额外用单独的语句来对变量和函数进行声明。

5.5.3 函数原型声明

单独对函数进行声明（英文动词为declare，名词为declaration）的方式是在程序中写出函数原型（function prototype）——或者说是“函数原型声明”。

函数原型在书写形式上与函数头部类似，只是在最后加一个分号，构成一条简单语句。

```
double scircle(double radius); ←
```

```
double srect (double a, double b); ←
```

```
void prtStar(); ←
```

原型声明中参数表里的参数名可缺（只写类型）。即使写参数名，所用名字也不必与函数定义用的名字一致。

```
double scircle(double); //只写参数类型，不写参数名
```

```
double srect (double length, double width); //参数名与定义中不同
```

函数原型声明可以出现在任何可以写定义的地方。目前人们认为最合理的方式，是把函数原型声明都放在源程序文件最前面。这样，本程序文件中所有的函数使用点都向上可以“看到”这里的原型说明。

函数原型的功能只是单独对函数进行声明，它并不能代替函数定义。程序中还是必须要写有函数定义才行。

只是因为函数原型能单独对函数进行声明，因而就可以更灵活地安排函数定义的书写位置：不再强求函数定义一定要写在主调函数前面，而是可以写在主调函数后面，或者写在另一个文件中。

➔ “先声明后使用，在别处作定义”。

5.5.4 外部变量的声明

外部变量是在函数外部定义的全局变量，它的**作用域**是从变量的定义处开始，到本程序文件的结尾。在此作用域内，全局变量可为各个函数所引用。

在多文件开发工作中，有时候需要**扩展**或限制外部变量的作用域。

假如在某个多文件开发的项目中有文件 fa.cpp 和 fb.cpp，它们中的函数都要用到同一个 int 类型的外部变量 g_num。解决的办法是仅在一个文件中作外部变量定义：

```
int g_num;
```

而在另一个文件中作一个外部变量声明。声明方法与变量定义类似，但要在最前面增加关键词 extern：

```
extern int g_num;
```


在多文件开发工作中，另一种可能的需求是限制某个外部变量的作用域仅在本文件中，而不能被其它文件引用。这时可以在源程序文件中用 static 声明或定义外部变量，使该变量的作用域限定在本文件中。

例如，在某源程序文件中有如下外部变量定义：

```
static int total;
```

则变量 total 被限定在该源程序文件中使用，而不能被其它文件所使用。

5.5.5 函数分解程序实例

【例5-25】猜数游戏程序：程序自动生成一个位于某范围里的随机数，要求用户猜这个数。用户输入一个数后，程序有三种应答：too big, too small, you win。

整个程序的工作流程基本设计：

从用户得到数的生成范围

do {

 生成一个数m

 交互式地要求用户猜数，直至用户猜到

} while(用户希望继续);

结束处理

在此我们把这个程序拆分为多个函数；在开发过程中先进行功能划分，写出多个函数的原型声明，并写出main函数（其中包含对这些函数的调用），最后再逐个写出这些函数的定义。

- 可以把取得工作范围和取得下一猜数值的工作分别定义为函数：

```
int getrange();
```

```
int getnumber(int limit);
```

其中 getrange 要求用户提供一个2到RAND_MAX的值，如果用户提供的值超出范围，应该要求重新输入。

而getnumber取得的猜测值也应该在给定范围内，否则也应提示用户重新输入。这里我们想给用户几次重新输入的机会，如果超过次数仍然不对，函数就返回一个负值，通知调用处出现了异常情况，使调用函数的程序段可以处理这种情况。

- 判断“用户希望继续”的部分也可以定义为函数：

```
int next();
```

将它定义为一个返回0或1值的函数，用于控制程序大循环的继续或者结束。

写出程序的主体部分：

```
#include <iostream>
```

```
#include <cstdlib>
```

```
#include <ctime>
```

```
using namespace std;
```

```
int getrange();
```

```
int getnumber(int limit);
```

```
int next();
```

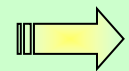
```
int main() {
```

```
    int max, unknown, guess;
```

```
    cout << "== Number-Guessing Game ==> endl;
```

```
    if ((max = getrange()) < 0) return 1; // 给定猜数范围。若  
    出错次数太多则退出
```

```
    srand(time(0));
```



```
do {  
    unknown = rand() % (max + 1);  
    cout << "\nA new random number generated. \n";  
    while (1) {  
        if ((guess = getnumber(max)) < 0) { //猜错次数太多则退出  
            cout << "Too many errors. Stop!";    return 2;  
        }  
        if (guess > unknown) cout << "Too big!\n";  
        else if (guess < unknown) cout << "Too small!\n";  
        else { cout << "Congratulation! You win!\n"; break; }  
    }  
} while (next());  
cout<< "Game over.\nThanks for playing!\n";  
return 0;  
}
```

接下来逐个考虑写出这几个函数的定义.....。

读入猜数上界的函数用常量限定用户出错次数，以免无穷循环。检查输入的合法性，合适时返回；有问题时要求用户重输。重复次数超过ERRORNUM时返回负值。

```
const int ERRORNUM = 5;
int getrange() {
    int i, m;
    for (i = 0; i < ERRORNUM; ++i) {
        cout << "Choose a range [0, max]. Input max: ";
        if ( !(cin >> m) || m < 2 || m > RAND_MAX) {
            cout << "Wrong number. Need a number in 2~" <<
            RAND_MAX << endl;
            cin.clear(); cin.sync(); cout << "Input again: ";
        } else
            return m;
    }
    return -1;
}
```

读入猜测数的函数与前一个类似。需要数值范围参数，检查有所不同，函数结构一样：

```
int getnumber(int m) {  
    int i, n;  
    for (i = 0; i < ERRORNUM; ++i) {  
        cout << "Your guess: ";  
        if (!(cin >> n) || n < 0 || n > m) {  
            cout << "Wrong number. Must be in 0~ " << m << ".\n";  
            cin.clear();    cin.sync();  
        } else  
            return n;  
    }  
    return -1;  
}
```

可以统一这两个定义。其中的提示串不同，下章考虑。

用户继续判断 next() 很简单:

```
int next() {  
    int ch;  
    cout << "Next game? (y/n): ";  
    while ((ch = toupper(cin.get())) != 'Y' && ch != 'N')  
        ; //空循环体  
    cin.clear(); cin.sync();  
    return (ch == 'Y' ? 1 : 0);  
}
```

把这些东西集成到一起，加上适当头文件就完成了。

改造：加入某些统计，输出统计数据。评价、策略等。

5.5.6 多文件开发实例

编写小型程序时，把所有内容写在单个文件中：单文件开发。

在较大的程序开发中，常常把程序分解为多个文件进行处理：多文件开发。

在多文件开发方式下，首先需要考虑如何把程序中的内容合理地划分并保存为多个文件。

按照惯例，程序的源文件分成两类，（1）包含实际程序代码的基本程序文件（*.c 或 *.cpp），称为**程序文件**或者**源代码文件**；（2）为上述基本程序文件提供必要信息的辅助性文件。

为了贯彻“使同一程序对象的定义点和所有使用点都能参照同一个描述”这一原则，人们常常将所有类型定义、常量定义以及各个函数的原型都列在一个或多个公用的信息文件里，然后让各个源程序文件都参看这个文件里的信息。

为此目的创建的文件信息的文件(*.h)称为头文件、head文件，或简称为h文件。

把程序中的内容合理地划分为多个文件之后，在编辑各个文件时需要协调好各个文件之间的变量与函数的声明、定义和使用。

【例5-26】以例5-25中的猜数游戏程序为例，以多文件开发方式编写该程序。

函数分解：

- 将用户继续判断定义为0/1值函数，控制大循环：

```
int next(void);
```

- 把取范围和取下一猜数定义为函数：

```
int getrange(void);
```

要求 2 到 RAND_MAX 的值，超范围就要求重输入。

- 把获得用户猜数定义为函数：

```
int getnumber(int limit);
```

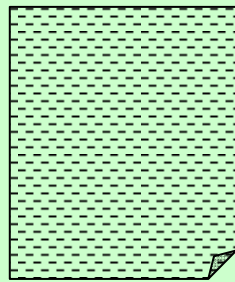
猜测值也应在范围内，否则提示重输。

给用户几次重输入机会，超过次数仍不对时返回负值，交给调用程序段处理。

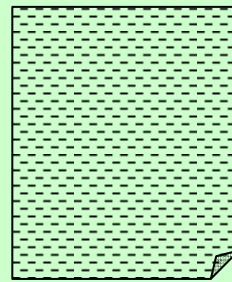
（这个程序并不复杂，可以把它们写在一个程序文件中。但我们以它为例来说明多文件开发）

按照多文件开发的一般规范，可以把基本信息保存在一个头文件中，把三个函数保存为一个源代码文件，主函数保存为一个源代码文件，总共三个文件（在实际的集成开发环境中，还需要一个项目文件，用于把这三个文件组织起来）。

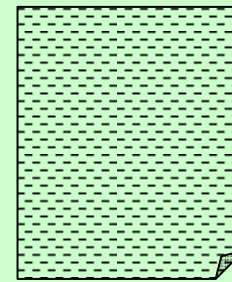
项目文件



guess.h



guess.cpp



main.cpp

创建一个名为guess.h的头文件：

```
#include <iostream>
```

```
#include <cstdlib>
```

```
#include <ctime>
```

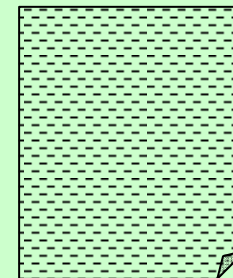
```
using namespace std;
```

```
int next();
```

```
int getrange();
```

```
int getnumber(int limit);
```

```
const int ERRORNUM = 5;
```



guess.h

把上例中的其它函数写在另一个文件中（guess.cpp）：

```
#include <iostream>
```

```
#include "guess.h"
```

```
extern const int ERRORNUM; //外部变量声明
```

```
int getrange() {
```

```
    .....
```

```
}
```

```
int getnumber(int m) {
```

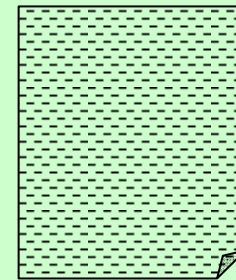
```
    .....
```

```
}
```

```
int next() {
```

```
    .....
```

```
}
```



guess.cpp

把 main函数专门写在一个文件中（main.cpp）：

```
#include "guess.h"
```

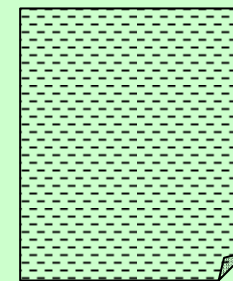
```
int main() {
```

```
.....
```

```
    cout<< "Game over.\nThanks for playing!\n";
```

```
    return 0;
```

```
}
```



main.cpp

这样就把这个程序的内容拆分保存在三个文件中。在后面两个源程序文件中，都写上了一句“`#include "guess.h"`”，其含义是把用户自行编写的名为“guess.h”的头文件包括进来。最后，在集成开发环境中建立一个工程（project，或译为项目）或解决方案（solution），把这三个文件包含进来，就可以对这个程序进行加工运行了。

从这个简单程序可以理解到多文件开发方式的一般规律。C/C++ 语言系统本身的实现也遵循这一方式。

一个C/C++ 语言系统总为我们提供了一组标准库头文件，还可能提供一些服务于特定系统（如DOS、Windows、UNIX等）的扩充头文件。这些头文件的作用就是为在程序里使用标准库函数以及其它功能提供必要的信息。

如果需要在程序里使用某些库函数，只要我们在源文件前面包含了必要的头文件，就能保证在编译过程能对源文件中有关函数调用正确进行处理。

还有一些情况下我们不希望将自己的源程序文件提供给别人。

上面的模块框架也使这种想法成为可能。假设我们不喜欢将 `guess.cpp` 提供给其他编程者，那么就可以只提供头文件 `guess.h`，再提供一个由 `guess.cpp` 编译后生成的目标文件（“`guess.obj`”）。其他编程者拿到这两个文件，就可以使用 `guess.cpp` 中的函数所提供的功能了。为此他们只要：

- 1、在编写程序时，让那些必要的源程序文件包含头文件 `guess.h`；
- 2、在连接时将我们提供的 `guess.cpp` 的目标文件也连接到可执行程序里。

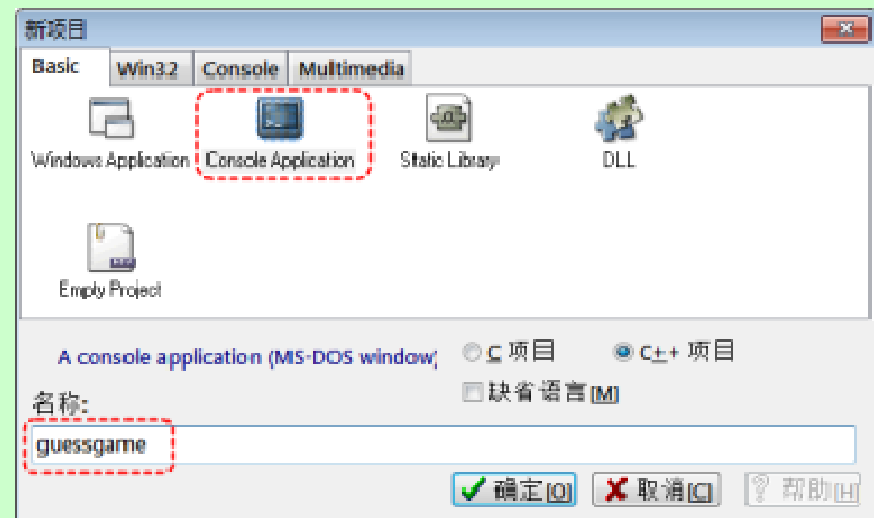
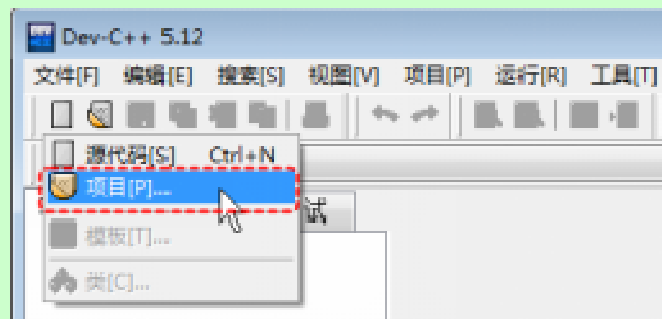
从这里可以看到标准库和其它程序库的影子。

Dev-C++多文件程序的开发实践

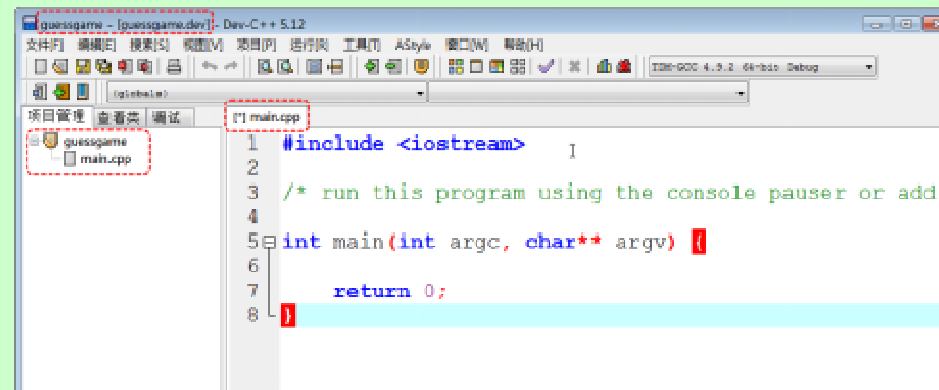
项目开发

在Dev-C++中进行多文件开发时，需要以“项目(project)”方式来组织和管理同一个项目中的程序文件。

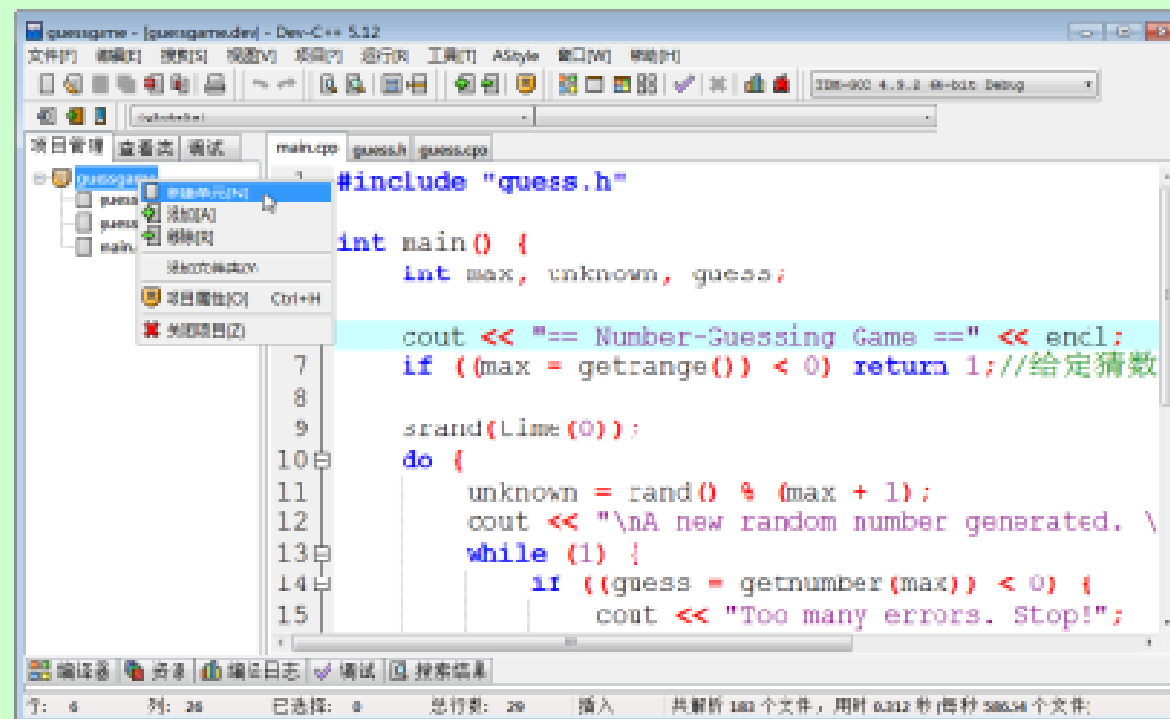
以“5.5.6 多文件开发实例”中的猜数游戏为例说明：



项目已建立:



可以新建单元、
添加和移除文件



项目的编译、运行等操作与单文件相似，在此不再重复。

项目所在的文件夹中除了 `guessgame.dev`、`guess.h`、`guess.cpp` 和 `main.cpp` 这几个程序文件之外，还有文件 `guessgame.layout` 和 `Makefile.win`，前者保存了该项目开发的工作状态参数，后者保存的是该项目进行编译加工时的参数。

当然，该文件夹下也可能会看到编译产生的目标文件 `guess.o` 和 `main.o`，以及编译生成的可执行文件 `guessgame.exe`。

第5章 函数与变量

5.1 函数的定义与调用

5.2 程序的函数分解

5.3 循环与递归

5.4 外部变量与静态局部变量

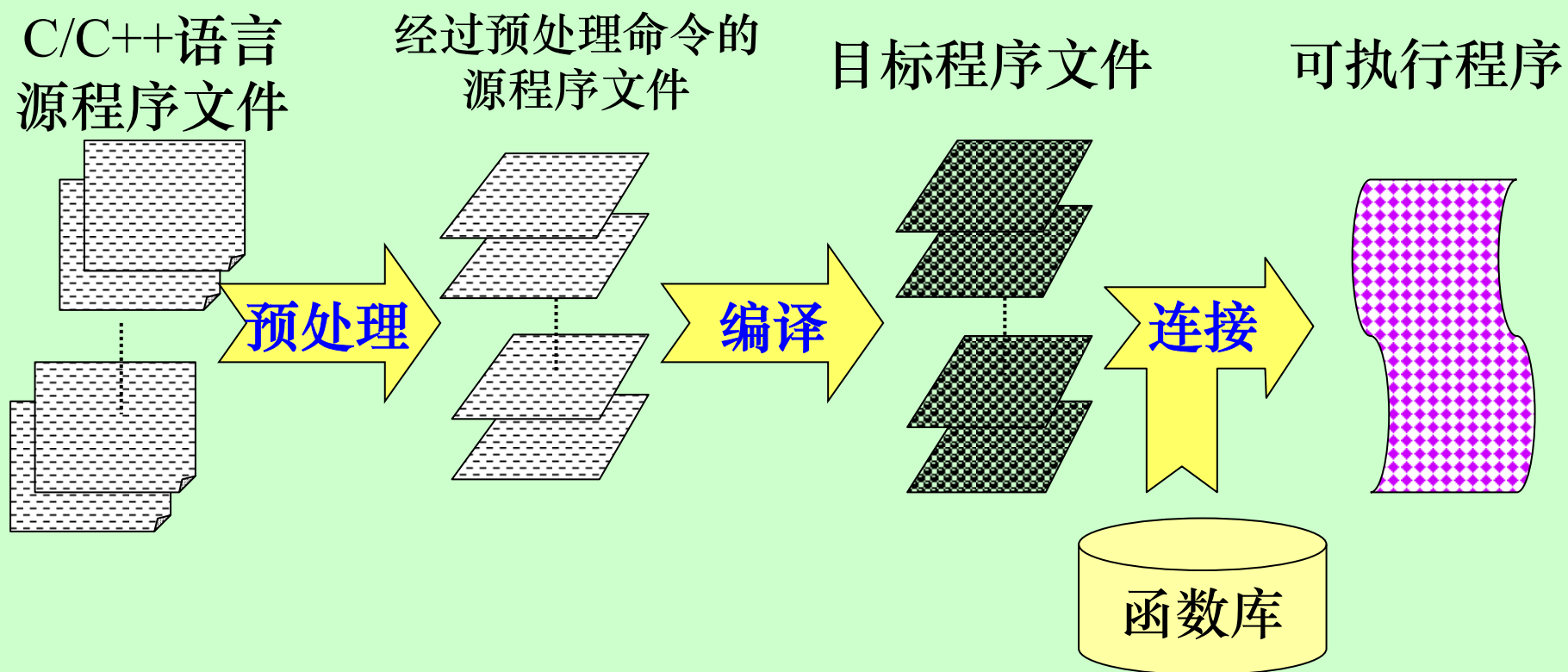
5.5 声明与定义

5.6 预处理

5.7 程序动态除错方法（二）

5.6 预处理

C/C++ 程序加工分为三步：预处理，编译，连接。



源程序的加工过程

预处理程序是 C/C++ 系统的一部分，处理源程序的
预处理命令行，产生修改后的源程序。

↓
第一个非空白字符是 # 的行



提供预处理命令是为了编程方便。

主要分为三类：

文件包含命令

宏定义与宏替换

条件保留命令（条件编译）

文件包含命令 把指定文件内容包含到当前源文件

- 形式1：#include <文件名>
用于包含系统头文件，预处理程序到指定目录找文件（通常指定几个系统文件目录）。
- 形式2：#include "文件名"
用于包含自己的文件。预处理程序先在源文件所在的目录里找，找不到时再到指定目录中去找。

处理过程：在文件系统中查找指定的文件，如果找到，就用找到的文件的内容取代该命令行。被包含文件里如有预处理行也会处理。

形式1：#include <文件名>

形式2：#include "文件名"

前面实例都用包含命令引进标准库头文件。

它们在系统子目录里（目录名为 include），内容是标准函数原型、系统使用的符号常量定义等。

包含这种文件相当于在源文件中写这些函数原型，使编译程序能正确完成对标准库函数调用的处理。

注意：写程序时一定要包含必要的系统头文件。

宏定义与宏替换

#define开始，两种形式：简单宏定义和带参数宏定义。

简单宏定义，形式：

#define 宏名字 替代正文

替代正文可以是任意正文序列，到换行为止。

如最后是“\”，下一行还作为宏定义的继续。

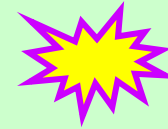
作用：为宏名字定义替代，由整个替代正文构成。

- 预处理程序记录宏名字及其替代。在源程序中遇到宏名字标识符时，就用替代正文替换。宏展开/宏替换。

替代正文里的宏名字还继续展开。字符串不做宏替换。

人们也用这种方式定义符号常量。

```
#define NUM 30
```



替代正文可以写任何东西。若定义：

```
#define SLD static long double
```

程序中的：

```
SLD x=2.4, y=9.16;
```

替换后变成：

```
static long double x=2.4, y=9.16;
```

```
#define NOSTOP while(1)
```

预处理程序做正文替换，替代正文可以是任何东西。

比较复杂而少用，初学者暂且跳过

带参数宏定义，形式：

#define 宏名字(参数列表) 替代正文

- 宏名字与括号间不能有空格，逗号分隔的标识符看作参数。替换正文为任意正文序列。
- 使用形式与函数调用类似，以类似参数的形式给出宏参数的替代段，用逗号分隔，称为宏调用。

```
#define min(A,B) ((A)<(B)?(A):(B))
```

```
z = min(x+y, x*y);
```

- 宏调用的替换分两步展开：先用各实参替代宏定义替代正文里的参数；再将代换的结果代入宏调用位置。

预处理中将被展开为：

```
z = ((x+y)<(x*y) ? (x+y) : (x*y));
```

使用带参宏与调用函数的意义不同。程序加工中在“当地”展开。程序执行中并没有调用动作，宏定义/调用中没有类型问题。一个宏能否使用/使用中发生什么/能否得到预期效果，完全看展开后的情况。

注意：宏展开可能引起参数多次计算。如：

`z = min(n++, m++);` 展开后的形式是：

`z = ((n++) < (m++)?(n++):(m++))`

替代正文各参数和整段应括起，避免出错。例：

`#define square(x) x * x`

在特定环境下可能出问题，例如：

`z = square(x + y);`

- 有些标准库“函数”用宏实现。getchar、putchar，ctype.h里字符类型判断。注意多次求值问题。
- 人们有时用宏定义简化程序书写。
- 带参宏的展开可避免函数调用开销，但使程序变长。
- 复杂宏定义展开后出错很难定位。

应谨慎使用（尽量少使用）宏。

- 写宏定义的常见错误是在定义行最后写分号。该分号将被代入程序，有可能引起语法错误。
- 宏定义从定义处起作用直到文件结束。一个文件里不允许对同一宏名字重复定义。#undef 取消已有定义：

#undef 宏名字

条件保留命令（条件编译）

`#if #else #elif #endif`

`#if/#elif`要求一个静态整型表达式

另两个单独成行

划出源程序中一些片段：

条件成立时保留，否则丢掉

根据条件成立与否从两段中取一段

根据多个条件决定从多段中取一段

条件应该是整数表达式，0表示条件不成立，否则条件成立。常用 `==`、`!=` 做判断。

例子：

```
#if TEST
    cout << ... ..;
#endif
```

谓词 `defined`。使用形式：

`defined 标识符` 或 `defined(标识符)`

当标识符是有定义的宏名字时，`defined(标识符)` 得到1，否则得0

`#ifdef 标识符` 相当于 `#if defined(标识符)`

`#ifndef 标识符` 相当于 `#if !defined(标识符)`

今天的上机练习内容：

- 例题程序 5-25， 5-26
- 编程练习题 5-10， 5-11

（把 5-11 的多个文件存放在一个名为“xxxx-5-11”的目录下，然后把整个目录打包，上传到QQ群文件中）

练习题5-11有点难，所以老师将会在 11:00 重新在腾讯会议中跟大家讲解。请大家及时参会。

第5章 函数与变量

5.1 函数的定义与调用

5.2 程序的函数分解

5.3 循环与递归

5.4 外部变量与静态局部变量

5.5 声明与定义

5.6 预处理

5.7 程序动态除错方法（二）

5.7 程序动态除错方法（二）

写好一个程序后，需要：

- 通过加工（编译和连接）产生可执行程序
- 运行它，提供数据进行试验，确认它确实满足要求
- 试验中常常会发现错误，需要设法排除

测试（testing）：在完成一个程序或一部分程序，通过编译后试验性运行，仔细检查运行效果，设法确认该程序确实完成了所期望的工作。反过来说：测试就是设法用一些特别选出的数据去挖掘出程序里的错误

排错（debugging）：发现程序有错时，设法确认产生错误的根源，修改程序，排除这些错误的工作过程

程序测试

- 测试时考虑的基本问题是提供什么样的数据，才可能最大限度地把程序中的缺陷和错误挖出来。

有两类确定测试数据的基本方式：

1. 根据程序结构确定测试数据。这相当于把程序打开，根据其内部结构考虑如何检查它，设法发现其中的问题。这种方式称为**白箱测试**。
2. 根据程序所解决的问题去确定测试过程和数据，不考虑程序内部如何解决问题。这相当于把程序看作解决问题的“黑箱”，因此称为**黑箱测试**。

在第3章中零散地讲过

- **白箱测试**：考察程序内部结构及由此产生的执行流，选择数据使程序在试验运行中能通过“所有”可能出现的执行流程。
 1. 复合结构只有一条执行流
 2. “if(条件) 语句1 else语句2”有两条可能执行流：条件成立时执行语句1；不成立时执行语句2。应设法提供测试数据，检验程序在这两种情况下都能正确工作。
 3. 从本质上说“while(条件) 循环体”可能产生无穷多条执行流：循环体不执行，执行1次，执行2次，……。无法穷尽检查。常用方法是选择测试数据，检查循环的一些典型情况，包括循环体执行0次、1次、2次的情况，以及一些其他情况
- 其他结构可类似分析。要考虑程序中结构嵌套产生的组合流程。

基本的策略是：

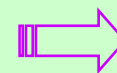
(1) 在写出程序之后，先借助自己的手和眼睛，查看一下程序对典型数据的工作情况，常能发现一些很表面的常见错误。

(2) 写一个测试程序，让程序自动地对更多数据执行测试。通常的办法是把程序的主体部分嵌入到一个更大的循环结构中，让某个循环变量依次发生变化，从而提供更多的测试数据。

在做黑箱测试时，我们考虑的不是程序的内部结构（也可以假设它并不可知，即使可以知道，也可能因为是别人做的，或者因为它极其复杂，因此理解其所有的执行流非常困难），而只是考虑程序所解决问题的各种情况。处理方法与做白箱测试是类似的。

使用调试工具除错

- 语法错误比较容易排错，第一章已经有简单介绍。
- 排错语义错误（逻辑错误）时，要设法确定错误根源，确定程序在什么执行流中产生错误，主要手段是选择适当的测试数据：
 1. 设法确认程序对最基本的情况能正常工作
 2. 解决了基本情况后再考虑更复杂的情况
 3. 设法找出出错的规律性，检查出错时数据经过的执行流，逐步缩小可疑范围。
 4. 在程序中加入输出语句，检查重要变量的值的变化情况
 5. 利用 IDE 的排错功能
- 测试程序、排除程序错误的最重要工具就是眼睛和头脑
- 程序的良好格式极其重要



- 上一章讲过了适合于初学者使用的两种排错方法：
整理排版缩进之后通读源代码
添加输出语句显示中间量的值
- 更复杂一点，就必须学会使用开发环境中的调试工具，逐步观察程序的运行过程，并观察变量的数值变化，加上自己的逻辑思考，才能找出程序中的错误所在并修改清除之。
- 下面介绍Dev-C++ 5.13 中文版中的调试工具。



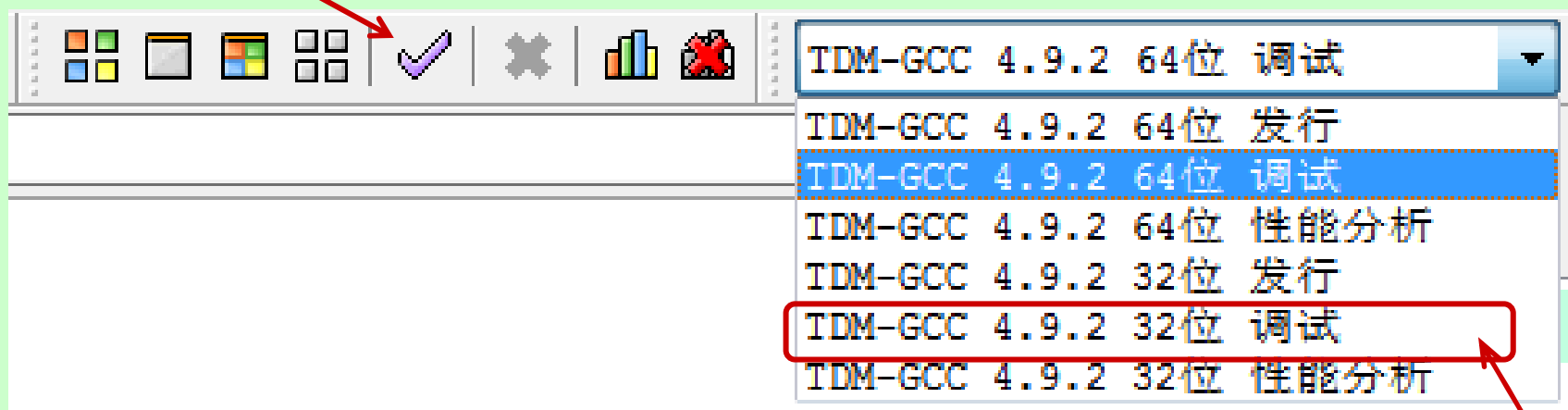
请下载安装老师提供的最新版本

Dev-C++ 5.13 中与调试有关的菜单项和工具按钮

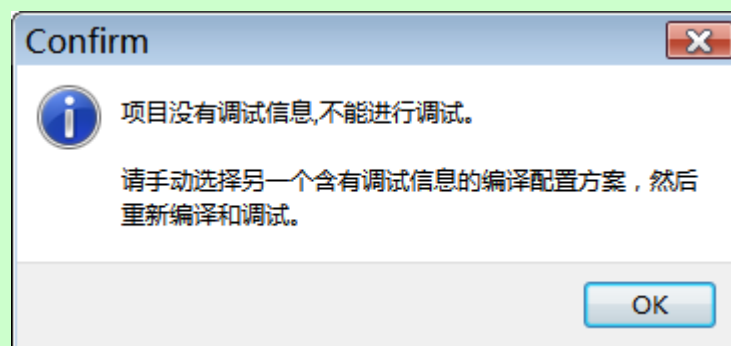


1. 开始调试 (Debug)

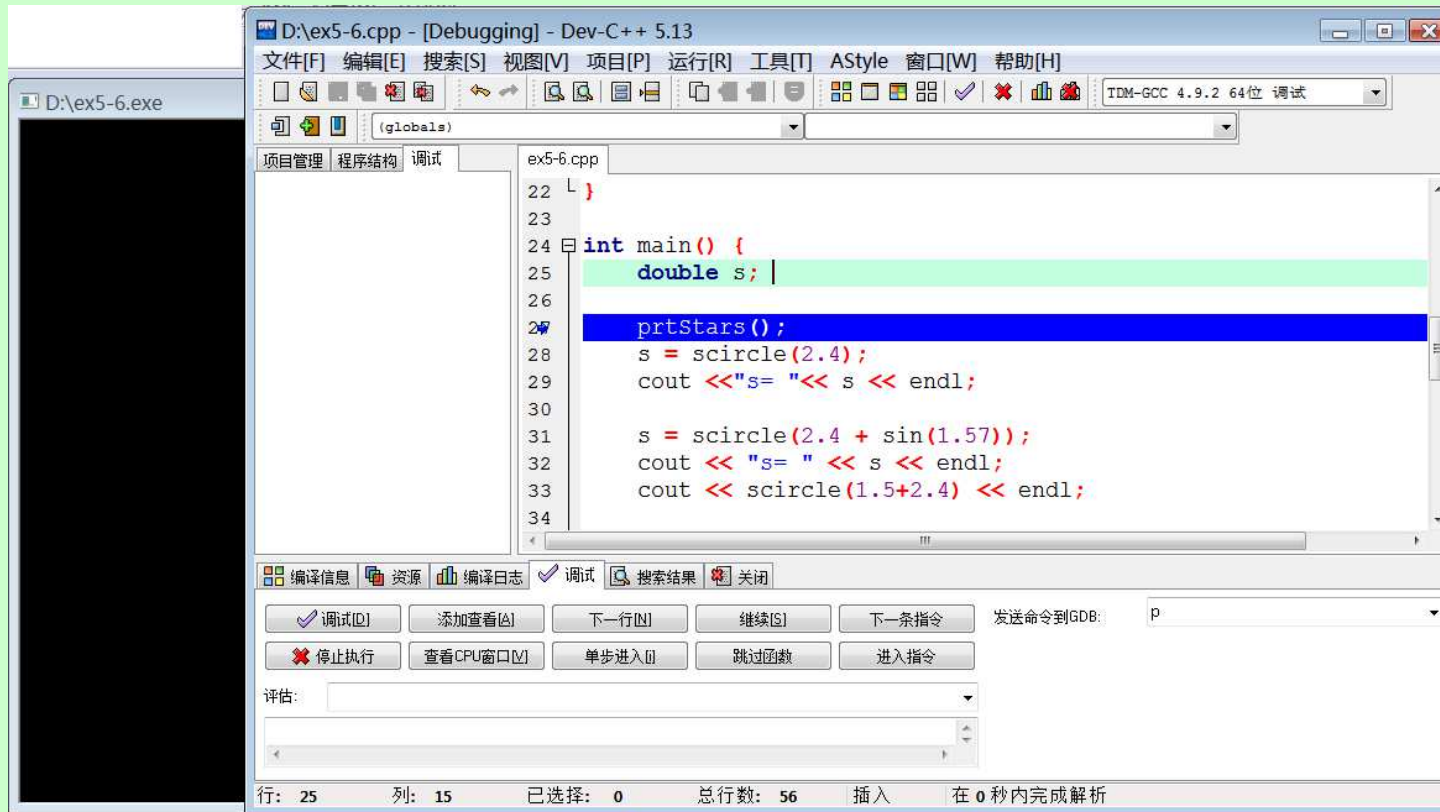
点击 调试(F5) 按钮，或按 **F5**，或点击菜单 “运行 -> 调试”



如果当前的编译器配置方案不含调试信息，则会弹出对话框，提示必须手工选择含有“调试”信息的编译器配置方案。



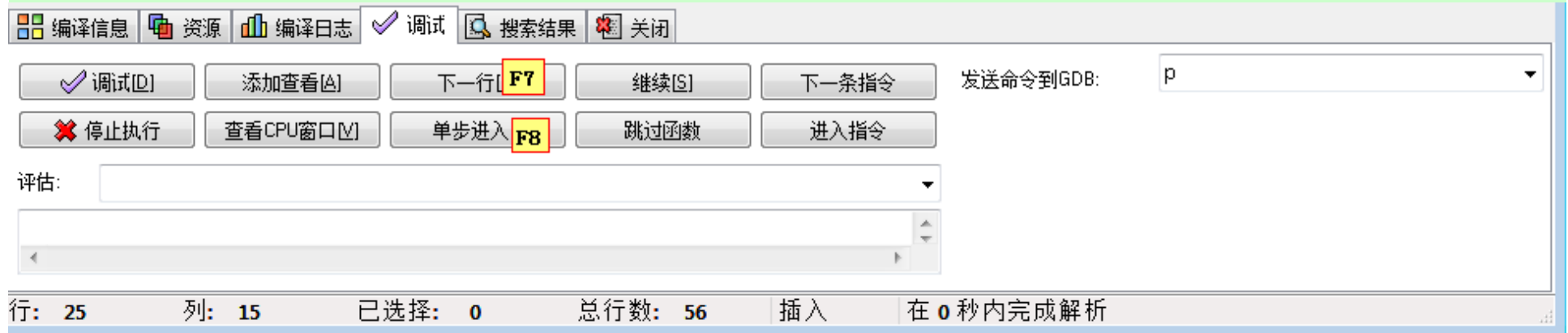
程序开始运行，到main 函数中第一条可执行的语句处暂停。



这时候最好手工调整一下 Dev-C++ 窗口的大小和位置，以便能够同时看到终端窗口和 Dev-C++ 窗口。

2、调试过程中的操作

自动显示调试面板，可以用鼠标点击按钮或按快捷键操作。



- “下一行[F7]”是指把当前语句作为一步执行完毕；
- 而“单步进入[F8]”是指如果当前语句中含有函数调用，则追踪进入到函数中去执行。
- 如果调用函数是标准函数或你认为无误的函数，就用“下一步”执行（以免追踪进入），对于怀疑有问题的函数才用“单步进入”去追踪。

3. 查看变量的值

调试过程中，常常有必要查看变量的值，了解其变化情况。

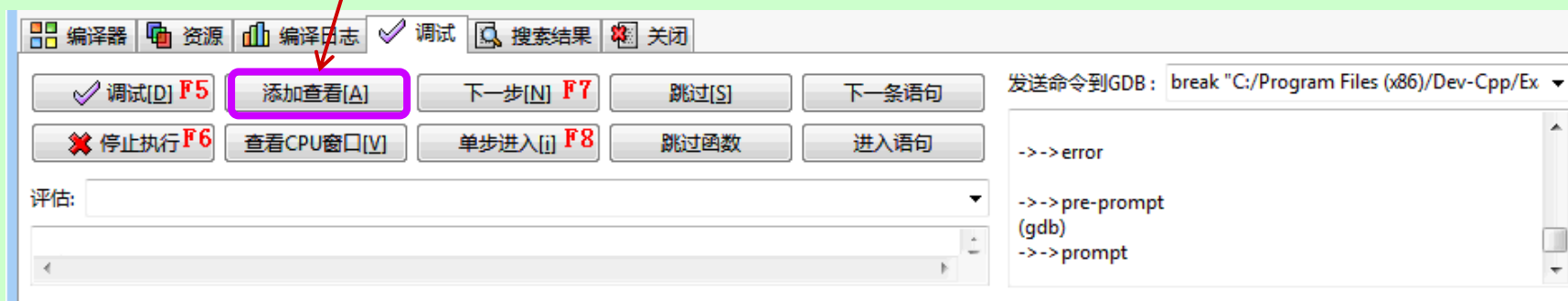
- 鼠标悬浮于变量上方，可看到当前的值。

```
priStars();  
s = scircle(2.4);  
cout << s << endl;
```

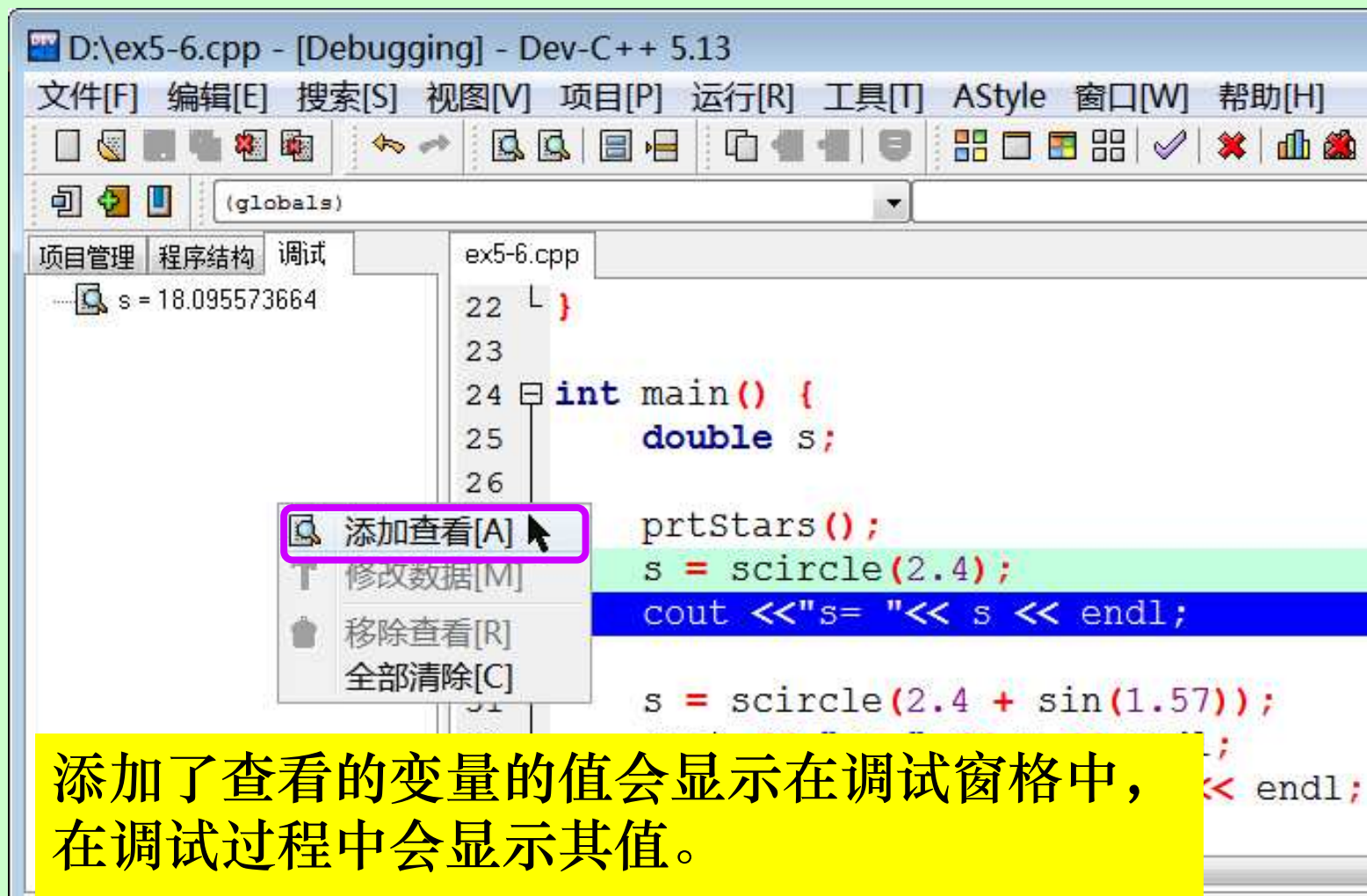
- 添加查看：

先在源程序中选中变量，
然后点击“添加查看”按钮；
所选变量就出现在“调试”窗格中，其值自动更新。

```
priStar();  
s = scircle(2.4);  
cout << "s= " << s << endl;
```



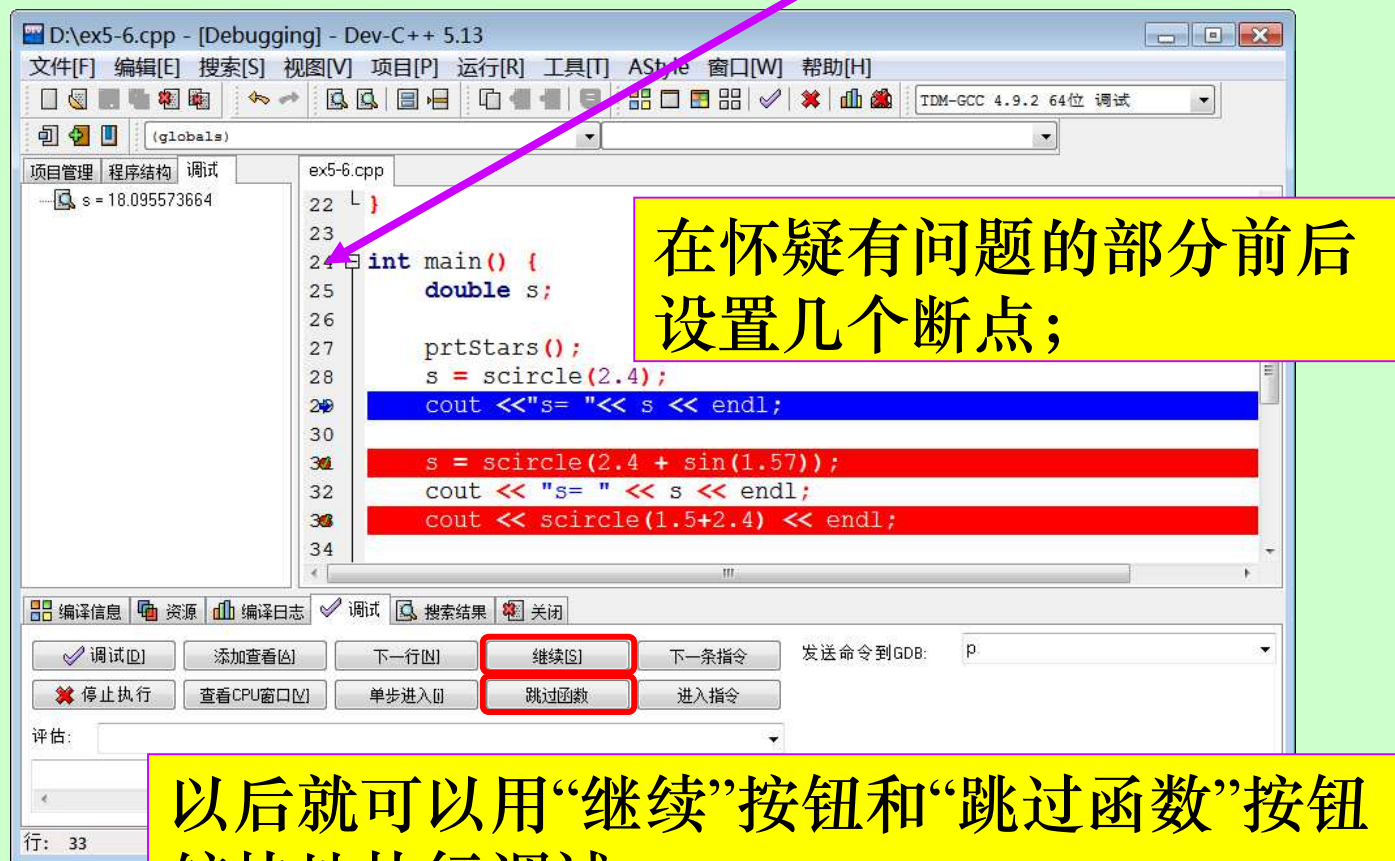
或者，在程序左边的调试窗格中点击鼠标右键，“添加查看”



添加了查看的变量的值会显示在调试窗格中，在调试过程中会显示其值。

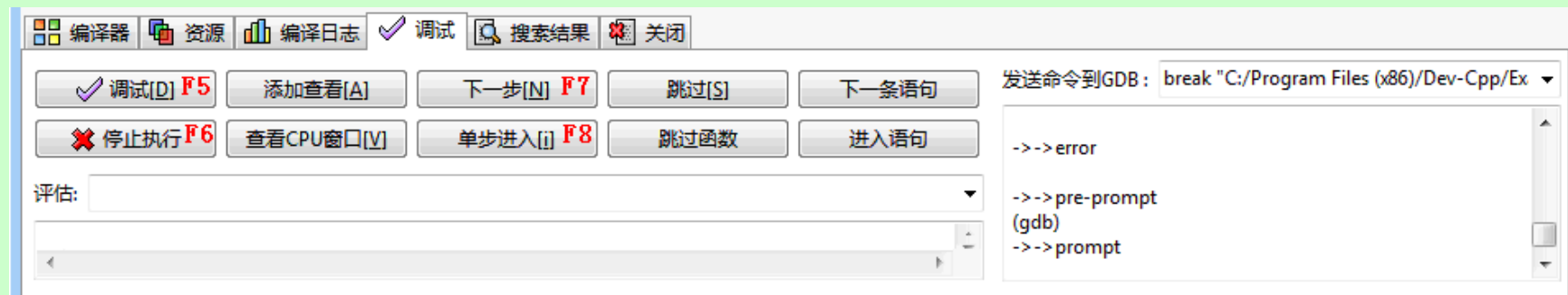
4. 设置断点 (Break point)

可以设置断点，以便快速地进行调试。
把光标移动到想要暂停执行的那一行，按 F4键，
或者用鼠标点击装订区位置中的行号。



调试 & 修改

- 灵活运用“下一步” (F7)和“单步进入” (F8)，并随时查看变量的值，在头脑中进行分析，从而判断程序中是否存在逻辑错误。
- 在调试中途或最后需要按“停止执行” (F6)以结束调试。然后根据调试过程中的思考结果对程序进行修改。
- 通常需要反复进行调试和修改才能排除程序中的错误。



以下为多余的重复页面

5.4.4 函数原型

- 命名对象（变量/函数等）有一个定义点，以及可能多个使用点。基本规则是先定义后使用。
- 保证正确编译的基本原则：从每个对象的每个使用点向前看，能得到使用该对象的完备信息。
- 局部变量：变量定义在语句前，保证了先定义后使用。后面变量定义可引用前面已经定义的变量。
- 函数：使用点所需信息就是函数的类型特征，包括函数名/参数个数和类型/返回值类型等。
- 调用位置要检查参数个数和类型，是否需要以及能否转换。对返回值也有类似问题。不知道函数的类型特征就无法保证正确完成这些检查和处理。

- 原型声明的形式与函数头部类似，加分号。参数名可省略，可与函数定义用的名字不同。



- 原型的参数名最好用有意义的名字，有利于写注释。

- 任何可以写定义的地方都可以写原型说明。提倡把原型说明都放在程序文件最前面：

1) 使文件里所有调用能看到同一个原型说明。

2) 使函数的定义点也能看到同一个原型说明。


原型说明是保证函数定义/使用间一致性的媒介。

为保证原型能起作用，必须给出完整的类型特征

```
void line(char c, int begin, int end);
```

```
void points(char c, int fst, int snd);
```

为避免函数使用的隐含错误，应坚持正确编程原则：

1. 如果使用标准库，**必须** #include 必要的库文件 
 2. 所有使用前未给出定义的函数（无论实际上在哪里定义），都**必须**给出正确完整的原型说明
 3. 应把原型说明写在源文件最前面
- 基本原则：使函数的定义点和所有使用点都能“看到”同一个原型说明。
 - 坚持这些原则，就能避免函数调用与定义不一致的错误（常常是编译程序检查不到的隐含致命错误）。

简单地说“用C语言编程序容易出错”并不合理。
许多错误是由于人们没按正确方式做事情。

5.5.1 外部定义的变量

- 在函数之外定义的变量称为外部变量/全局变量。
- 外部变量原则上可在程序中任何地方用。由于变量定义从出现处开始起作用，通常把外部变量定义写在源文件最前面，使文件里的函数都可以访问它们。

```
#include <stdio.h>
int n; //外部变量
int isprime() {
    for (int m=2 ; m * m <= n; m++) if (n % m == 0) return 0;
    return 1;
}
int main() {
    for (n=2; n<=100; n++) if (isprime()) cout << n << " ";
    return 0;
}
```

- 外部变量可以后定义先使用，或在一个源文件里定义在其他文件使用。为此，使用前应给出外部变量声明。形式与变量定义类似，前面加关键词extern。
例：extern int n, m;
- 外部声明通常放在源文件最前面，供整个文件参照。
更重要的：保证整个程序参照同一声明，保证一致性。

```
#include <stdio.h>
extern int n; //外部变量声明
int isprime() {
    for (int m=2 ; m * m <= n; m++) if (n % m == 0) return 0;
    return 1;
}
int main() {
    for (n=2; n<=100; n++) if (isprime()) cout << n << " ";
    return 0;
}
int n; //外部变量定义
```



- 书上例。后面会看到有更有意义的例子。
- 新情况：如果一个函数定义里用到外部变量，它就依赖这些变量，不再独立了。

两点注意：



- 定义和声明不同。定义要求创建被定义的对象；声明只指明其存在，必须另有定义，否则该说明无效。（有关变量定义与说明的差别下面还要讨论。）
- 外部变量可在整个程序用，在一个完整程序里不能有重名外部变量。否则连接时会出问题
- 不要与库里东西重名（标准库定义了一批外部名字）。

5.5.2 作用域与生存期

- 变量定义：1) 定义特定类型的变量；2) 为变量命名。还确定了：
- 变量定义起作用的范围，**变量定义的作用域**。由定义位置确定，在此范围可通过该名字使用该变量。
- 确定变量建立和销毁时间，**变量的存在期**。各种变量的存在期可能不同。变量实现的基础是内存单元，存在期就是变量被**分配内存存储到撤消**的期间。
- 作用域/存在期是重要概念。有联系但又不同。弄清它们，许多问题就容易理解了。

- 一个定义的作用域是一段源程序，是静态概念。如在函数体开始定义的变量，作用域是整个函数体。
- 存在期是动态概念（程序执行的一段时间）。变量在存在期中保持其存储单元，不经赋值那里的值就不变。
- 在作用域和存在期方面，外部变量和函数内的普通局部变量（称为自动变量）性质截然不同。
- 外部变量定义的作用域是整个程序（全局的），这样定义的变量可以在程序中任何地方使用。
- 自动变量定义的作用域是定义所在的复合语句。在该复合语句之外无效（局部的）。
- 形参看作函数体的局部变量，作用域是整个函数体。

外部变量和自动变量

- 从作用域角度看，main也是普通函数，其中的定义是局部定义，作用域是main函数的体。
- 外部变量的存在期是程序整个执行期间。程序开始时建立所有外部变量，一直延续到程序结束。
- 复合语句里定义的变量，存在期是该复合语句的一次执行。复合语句开始时创建，结束时撤消。复合语句再次执行时重建，新变量与撤消的变量无关。自动变量。

```
for (n = 0; n < 10; n++) {  
    int m;  
    if (n == 0) m = 2;  
    /* 循环第二次到这里时m的值未定 */  
}
```

两种主要作用域

- **全局作用域**是所有外部定义（外部变量定义/函数定义等）的作用域。
- 每个复合语句确定一个**局部作用域**。
- 还有一种以源程序文件为单位的作用域（后文）

```
#include <stdio.h>
int n;    //外部变量
```

全局作用域、局部作用域可以嵌套

```
int isprime() {
    for (int m=2 ; m * m <= n; m++) if (n % m == 0) return 0;
    return 1;
}

int main() {
    for (n=2; n<=100; n++) if (isprime()) cout << n << " ";
    return 0;
}
```

- 变量的作用域与“是否可以使用”是两回事。——必须“先定义再使用”或“先说明(后定义)再使用”。

例如，外部变量定义之前的代码中无法用它。

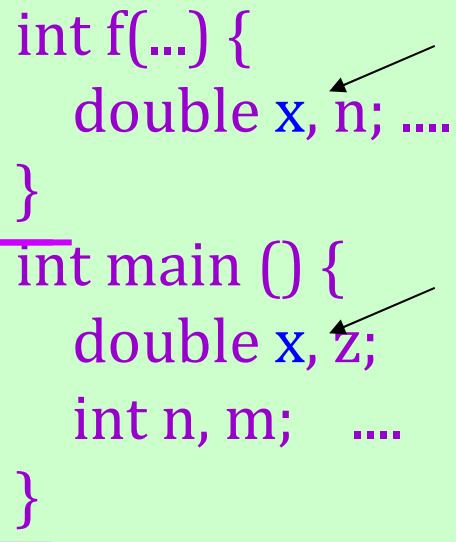
```
int main () {  
    /* 这里无法用num */  
}  
  
int num;  
  
int f(...) { ... num ... }
```

可把变量定义前移，或另写外部变量说明。

变量定义的嵌套

- 函数定义只出现在全局作用域中，不可能嵌套。
- 变量定义可出现在复合语句里，可能出现嵌套定义。
- 规定：同一作用域里不允许定义多个同名变量。
- 不同作用域中可以定义同名变量。不同作用域里定义的同名变量互不相干。例：

```
int f(...) {  
    double x, n; ....  
}  
int main () {  
    double x, z;  
    int n, m; ....  
}
```



嵌套作用域里可能出现同名变量定义，例：

```
int f(int n) {  
    int x, y;  
    ....  
    while (....) {  
        double x;  
        ....  
    }  
    ....  
}
```

- 1、不同作用域定义的重名变量互不相干。
- 2、新规定：内层作用域里的同名变量定义（在此局部）遮蔽外层同名变量定义。

常变量

- 用关键字 `const` 定义的变量。是变量，但不允许赋值，只能初始化，其存在期中总代表同一个值：

```
const int num = 10;
```

- 若出现在局部作用域里，那么也动态建立和初始化，每次初始化的值可以不同：

```
for (i = 2; i <= 200; i += 2) {  
    const int n = i * i; ... ..  
}
```

可用 `const` 定义常参数，函数体里不能重新赋值。

许多程序里用常指针参数（后面讨论）。

5.5.3 变量的其它问题

- **寄存器变量**。自动变量可加关键字register，定义为寄存器变量。实际安排由编译器决定。寄存器变量无地址，不能做地址操作（指针操作，第七章）。
- **外部静态变量**。全局变量作用域太大，容易出现名字冲突。静态外部变量是作用域为源文件的外部变量。静态外部变量用关键字static说明。
- **静态函数**：在函数的返回类型前加关键字static，只能在一个源文件内用。不同文件里定义的同名静态函数不会相互干扰。
- 这里的静态与函数局部变量的“静态”意义不同。

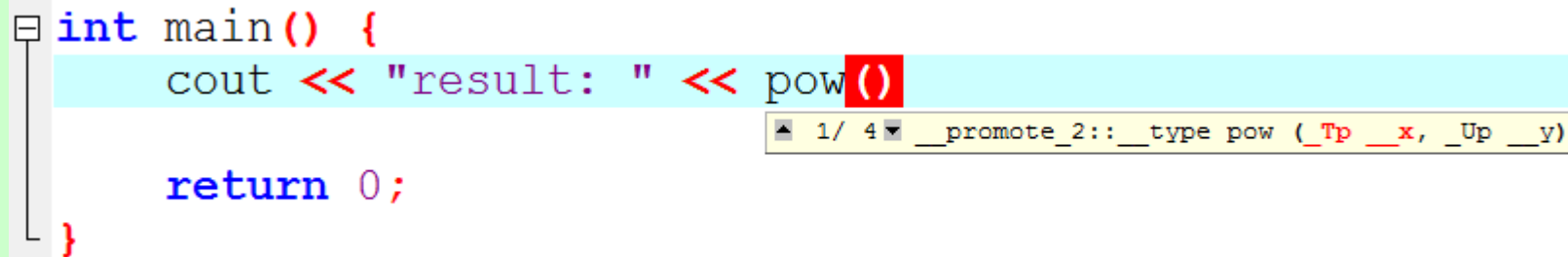
前五章总结

- 讨论了基本的程序和程序设计问题
- 源程序，加工和执行；
- 数据，类型和表达式计算；
- 执行控制，基本控制结构和执行流程；
- 函数定义与调用，程序功能分解；
- 数据内部与外部形式之间的转换，输入和输出

Dev-C++多文件程序的开发实践

函数提示

在一个源代码文件中包含了必要的库文件并保存之后，如果继续编辑时键入了函数名称和圆括号，则 Dev-C++ 会自动弹出该函数的类型特征说明，以帮助用户正确地输入实参。



```
int main() {  
    cout << "result: " << pow()  
    return 0;  
}
```

1/ 4 __promote_2::__type pow (_Tp __x, _Up __y)